Accepted for Publication at:
2nd Workshop on SHAring and Reusing architectural Knowledge - Architecture, rationale, and Design Intent (SHARK/ADI) at 29th International Conference on Software Engineering (ICSE 2007).



# Exploring Extensibility of Architectural Design Decisions

Salvador Trujillo, Maider Azanza, Oscar Díaz, Rafael Capilla
The University of the Basque Country / Universidad Rey Juan Carlos
San Sebastian / Madrid, Spain
{struji,maider_azanza,oscar.diaz}@ehu.es;rafael.capilla@urjc.es

***Abstract*— Software architectures represent the design of a system for describing its main relevant parts. Recently, recording and documenting architectural design decisions has attracted the attention of the software architecture community. Design decisions are an important piece during the architecting process that must be explicitly documented, but there is little evidence of successful reuse of this architectural knowledge. This work focuses on the reuse of design decisions in order to customize architectures. Specifically, we explore extensibility ideas from software product lines to show how architectures can be extended on the basis of design decisions. The documentation of synthesis architectures has received so far little attention, and particularly its reuse. This ongoing research describes an approach for product line synthesis architecture, where design decisions are introduced to promote its reuse.**

## I. Introduction

Software architectures represent the design for describing the main parts of a software system. Traditionally, software architectures have been considered as a set of interrelated components and connectors [2], [19]. According to this definition, the functionality of a software system is mainly described by means of a set of interrelated components and connectors [13], [14].

Prior work [18] described the rationale and principles that guide the design and evolution of software architectures. This rationale is considered when adopting an architecture-centric approach. However, the main goal for representing architectural design decisions is to bridge the gap between software requirements and architectural products.

The need for recording, documenting and managing design decisions has been recognized in recent workshops and conferences [1], [15]. The fact that design decisions are never recorded complicates architecture reconstruction. This difficulty to recreate lost or non documented decisions is one of the main reasons to record them. Hence, documenting architectural knowledge enables not only to track the overall architecture along the construction process, but also to support future maintenance and evolution activities.

This paper focuses on software product line synthesis (a.k.a., product derivation), which defines the process to synthesize individual systems (e.g. how artifacts are assembled to yield an outcome product) [23]. More to the point, the interest rests on the abstraction of this synthesis process in terms of abstract models representing specifically the parts of product-line synthesis [21]. As such synthesis abstraction represents the parts that form the synthesis process, it is designated as *synthesis architecture*.

So far, reuse on such architectures was limited to generative techniques, where modifications in the abstractions implies a re-generation of its implementation (i.e, taking synthesis abstraction as input, a transformation generates an output synthesis script). This kind of reuse enables to customize synthesis with little effort when needed (e.g. to vary funcionality of resulting scripts).

However, the modification of these abstractions is not currently being documented. Doing so, the modifications in the design decisions that might impact on the architecture are not tracked down. Therefore, this work explores how design decisions can be documented, and how they affect the synthesis architecture. We show the extensibility of architectures based on certain design decisions, and we illustrate our ideas with an example, which is elaborated on the benefits gained from the combination of architectural knowledge with our work.

The remainder of this paper is as follows. Section II describes existing background for synthesizing product line programs. Section III addresses the problem statement which can be ameliorated using architectural knowledge. Section IV deals with architecture extensibility, and how design decisions are used for improving some key aspects of product synthesis. Section V connects with related work and section VI concludes.

## II. Background

*Model Driven Development* (MDD) is an emerging paradigm for software construction that uses models to specify systems, and model transformations to synthesize executables [10]. *Feature Oriented Programming* (FOP) is a paradigm for software product lines where customized programs are synthesized by composing features [5]. *Feature Oriented Model Driven Development* (FOMDD) is a blend of FOP and MDD that shows how products in a software product line can be synthesized in an MDD way by composing features to create models, and then transforming these models into executables [23]. This work focuses on how systems are synthesized in FOMDD.

FOMDD supports synthesis using scripting, which when executed, synthesizes individual systems. The creation of this scripting involves repetitive, time-consuming and cumbersome tasks. However, such architectures are not monolithic blocks,

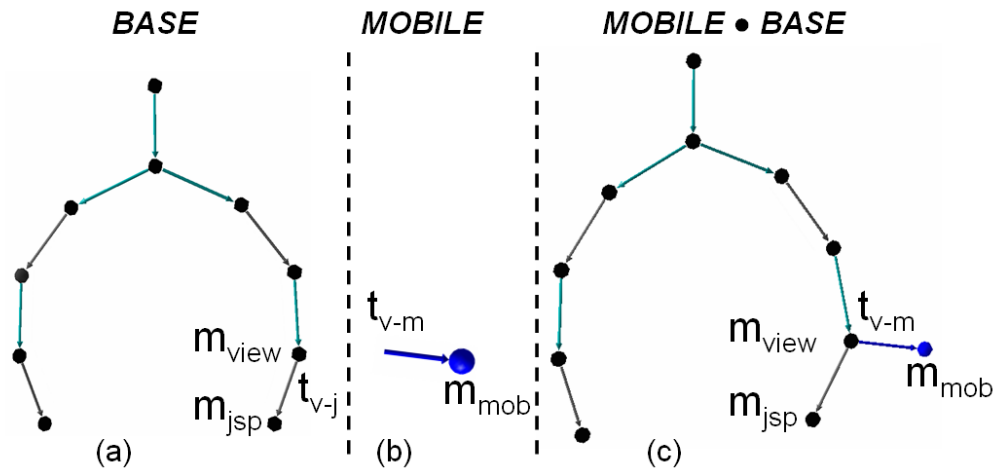


Fig. 1. Extension of Synthesis Architecture

but consist of distinct parts representing different types of design decisions (e.g. operations to synthesize a program). Recently, the creation of scripting was simplified by introducing an approach to specify an abstraction from which synthesis scripting can be generated (raising the abstraction level) [21].

Figure 1a shows an example where the representation of distinct parts completes a synthesis architecture. Note that nodes represent models (e.g. $m_{view}$ and $m_{jsp}$) and arrows represent model transformations (e.g. $t_{v-j}$ transforms from input model $m_{view}$ to output $m_{jsp}$). Refer to [21] for additional details about the example. This paper focuses on this synthesis architecture, which is designated as *architectural metaprogramming* by Batory [3]. In the next section, we explore synthesis architectures in relationship to architectural knowledge.

## III. Problem Statement

In general, software architecture plays an increasingly important role to manage the complex dependencies and interactions between stakeholders [15]. Existing approaches for documenting software architectures typically focus on the description of components and connectors, but fail to reflect the decisions made along the architecting phase. During the evolution of any software system, architecture erosion may cause high maintenance costs because the decisions made in the past were not documented, and this architectural knowledge is vaporized.

This situation is faced by documenting *Architectural Knowledge* (AK), in such a way that the decisions made can be recovered and recreated during any maintenance process. AK is defined as the integrated representation of the software architecture of a software-intensive system (or a family of systems), the architectural design decisions, and the external context/environment [15]. According to [12], an architectural design decision is "*a description of the set of architectural additions, substractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements that (partially) realize one or more requirements of a given architecture*".

Other approaches propose models and tool support for recording and documenting architectural design decisions [6], [7], [12]. Although these approaches record decisions, they provide limited support for the discovery and reuse of AK. The documentation of design decisions constitutes a first step towards AK reuse, but documentation alone does not imply reuse.

Because a synthesis architecture involves a set of (synthesis-specific) design decisions, in this work we try to apply the ideas for documenting design decisions during the transformations that happen along the synthesis. We advocate the use of *step-wise refinement* [5], [9] to extend synthesis architectures and their design decisions. Doing so, we expect *(i)* the relationships between a design decision and its impact on the architecture can be documented, so *(ii)* it can be easily tracked down, and most important *(iii)* being architecture extensible, it is possible to extend the base architecture (with base design decisions) with new decisions that extend such base architecture (with new features). Such extensibility is introduced next into the synthesis architecture in order to achieve expected benefits.

## IV. Architecture Extensibility

### A. Synthesis Architecture

Synthesis embraces the realization of several steps to yield an individual system (out of a product line). The synthesis architecture is the abstraction of such steps. A typical example consists of distinct transformations to synthesize executables from input models.

The point to stress is not the detail of the synthesis process itself (i.e. actual synthesis operations), but the decisions reflected in such synthesis (i.e. why such operation was decided). More to the point, such decisions that are not usually documented (more shortly). Synthesis-specific decisions affect to the expressiveness of our models, the target platform used, and the functionality of the code generated among others. Such decisions shape the architecture of Figure 1a, and the resulting


Fig. 2. Refinement of Architectural Documentation

architecture would depend on the early decisions made in the base architecture.

Currently, we have tools able to generate a script starting from a synthesis abstraction (Figure 1a). Such synthesis consists of a set of design decisions that describe its architecture [21]. When these decisions vary, the architecture is then customized, and it is possible to regenerate these scripts. The design decisions that are part of the base architecture (common to the entire product line) can be extended with new decisions during the transformation process. However, we do not keep track of these transformations so far, and in our context we are unaware of what is common and what is variable.

### B. Extensibility

The base architecture can be extended to add the necessary extensions to obtain the customized architecture ultimately. To attain this extensibility, the concept of *refinement* is used [5]. Synthesis architecture extensions can be achieved in terms of refinements. Inspired on Dijkstra's *Step-Wise Refinement* [9], a refinement is used to denote an increment in functionality [5]. The refinement of models is not new and prior work showed *statechart* model refinement [23]. In our work, extensibility implies that the base architecture contains a predefined list of extension points (similar, but not equal to variation points) that are used to extend the architecture.

Following this approach, we suggest to compose design decisions during the synthesis process. This enables us to record not only the decisions that may affect the base architecture, but also those related to the extensibility that yields the customized architecture. We first define a set of design decisions which are common to the base architecture (e.g. the set of models and their transformations from which program executables can be derived). Then, we define extension points on this base architecture to make the architecture extensible for new requirements.

In our approach, extensions deal with assorted design decisions. For instance, we can extend our synthesis architecture to support a new platform (e.g. from J2EE to .NET), to generate executable code for error-handling management, to improve the accessibility level of web pages, or to modify such web pages in order to serve them customized to mobile devices (e.g. generate WML in addition to HTML). The refinement of Figure 1b extends the base architecture of Figure 1a to support the decision to support mobile devices. As a result, a new node (e.g. $m_{mob}$) will appear in the architecture and a new arrow (e.g. $t_{v-m}$) connecting such node with an existing node has to be defined (note that this arrow contains as well the newly introduced design decisions). Figure 1c shows a resulting example where Figure 1a is extended with Figure 1b.

However, the idea is not only to document extensions, but also to support reuse through the composition of extensions. Next subsection illustrates the details of architecture composition and how reuse is achieved by composing the extensions in the architecture.

### C. Architecture Composition

*Graph eXchange Language* (GXL) metamodel is designed to specify directed graphs [11]. GXL was selected to depict the synthesis architectures because it provides a meta-model,

a language, and tool support [11], [20], which fits for the representation of synthesis. Figure 2a shows a fragment of GXL code that specifies the base synthesis of Figure 1a. This code shows the representation of a model conforming to the GXL meta-model. Note that this code also shows design decisions, which conform to the vocabulary introduced shortly in Figure 3.

We represent the synthesis architecture using an XML-based language (GXL) and we use XAK to specify the extensions made to the architecture. XAK supports the refinement of XML documents for product-line extensibility [4], [22]. Figure 2b shows a refinement example extending the previous architecture (according to Figure 1b where a new transformation was included to deal with mobile devices). Note that Figure 2a defines an extension point beforehand (see *xak:module="mGraph"*).

The use of GXL combined with XAK enables not only to represent synthesis architectures, but as well to compose a representation of the base architecture (Figure 2a) with its extensions (Figure 2b) to obtain the final architecture (Figure 2c). This composition is supported by AHEAD tooling [5].

Our aim is to compose any number of extensions to produce a customized synthesis architecture. We document as well the design decisions (more shortly) that make the composition of the synthesis architecture, from which a synthesis script is generated. Such script synthesizes the end product.

### D. Documenting Design Decisions

From our viewpoint, the composition of architectural extensions is not enough for several reasons. For instance, traceability between requirements and products can be better detailed by documenting the design decisions taken. Also, reuse in terms of architecture extensibility can be achieved by composing design decisions. Therefore, we advocate to document base design decisions and alternative and optional decisions that are used during the composition of the extensible points in the architecture.

To attain this, we document in each architectural element those decisions that directly impact its design. We specify the attributes of architectural design decisions that are useful to be used in the future. Note that Figure 2 represents design decisions (see dotted-lines and element enclosed by <*dd:decision*>). We introduce an XML-based language for describing the attributes of the decisions (see Figure 3). Most of these attributes have been used from the template described by Tyree and Akerman [24]. We have used those attributes we believe are relevant for the synthesis process.

Doing so, we document explicitly the relevant information that motivates the composition rules and extension points used. That is, Figure 2a, Figure 2b, and Figure 2c contain such documentation. We are not only extending the synthesis architecture, but also the design decisions that each architecture extension implies.

Embedding design decisions into synthesis obscures the resulting documentation (i.e. the documentation is included within synthesis abstraction). Nonetheless, it is possible to extract those design decisions from the GXL representation and transform them to a user-friendly HTML representation. To attain this, a trivial XSL transformation is only needed. Note that each customized synthesis architecture would have a different customized documentation of its design decisions.

### E. Traceability and Design Decisions

Each extension to the synthesis architecture encompasses not only the impact of synthesis-specific design decisions, but also the rationale behind them. Thus, the traceability of extensions and their architectural design decisions is immediate because we can trace how each decision extends the architecture. This is not only due to the use of refinements, but to the use of AHEAD, which structures product-line artifacts for the synthesis of end programs [5]. Alternatively, we use AHEAD in this work to structure design decisions associated to the synthesis. Hence, AHEAD traceability benefits are applied in this context [5].

## V. Related Work

Clements et al. modernize the description of architectural views in order to produce a more complete and accurate architecture documentation [7]. The authors suggest the need to document the rationale of the design decisions as part of the information required when documenting software architectures [7]. This approach provides an initial set of guidelines for documenting architectural design decisions.

Tyree and Akerman motivate the need to consider architecture design decisions as the way to understand the impact of the choices made by software architects [24]. The authors focus on the process by which design decisions are carried out and they provide a list of potential viable decisions. A template with attributes to characterize the design decisions is also described for documentation purposes.

Jansen and Bosch propose a metamodel to describe architectural design decision, which is supported by the Archium prototype tool [12]. The proposed metamodel consists of three sub-models: an architectural model, a decision model, and a composition model. In Archium, a software architecture is seen as a composition of design decisions, and the composition model is responsible of relating the changes of the decision model to the elements of the architectural model.

Capilla et al. provides initial support (model and prototype tool) to record architectural design decisions [6]. ADDSS is the web-based tool proposed to store and manage architectural design decisions [6]. Documented decisions can be used by different stakeholders but no specific reuse or discovering methods are proposed. As mentioned in [17], some knowledge management tools exist, but with little usage in the software architecture field. Diaz et al. tackle the variability of synthesis to deal with managerial strategies [8]. This work is the precursor to address such variability of synthesis at a higher abstraction level based on design decisions.

## VI. Conclusions

This ongoing work addresses the reuse of architectural design decisions. We have explored the extensibility of synthesis architectures, described an illustrative example, and reported

| Decision Attribute | Description | XML Representation |
|---|---|---|
| Decision ID | Identifier for the decision | <dd:decision id=" "> |
| Decision Issue | The architectural issue being addressed | <dd:issue> |
| Decision Name | Name of the decision | <dd:name> |
| Decision Description | Description of the decision | <dd:description> |
| Status (pending, rejected, approved, obsolete) | Denotes the status of the decision | <dd:status value={pending\|rejected\|approved\|obsolete}/> |
| Constraints | Restrictions that apply to a decision | <dd:constraint>@list |
| Alternatives | List of alternative decisions considered during the decisión making activity | <dd:alternatives>@list |
| Related Decisions | Show related decisions but complex dependencies between decisions can be defined here | <dd:dependencies>@list |
| Related Requirements | Requirements that motivated the decisions | <dd:req>@list |
| Related Artifacts | Link to architectural artifacts impacted by a decision | <dd:artifacts>@list |

Fig. 3. Representation of Design Decision

on the benefits of this approach. Our exploratory approach reported an initial attempt towards the reuse of synthesis architectural knowledge and design rationale.

This work promotes the traceability between architectural design decisions and architectural solutions. The next step of our approach is to explicitly connect requirements (in terms of features) to architectures via design decisions, so we can establish explicit traces between them. Future work should address this relationship between decisions and features (similarly to [16]). Doing so, dependencies between design decisions can be handled in terms of features and impact of change in terms of refinements.

Although not addressed specifically, we believe the evolution of architectural knowledge could be handled by the extensibility approach described in this work.

*Acknowledgments:* This work was co-supported by the Spanish Ministry of Science & Education, and the European Social Fund under contract TIC2005-05610. Trujillo has a doctoral grant from the Spanish Ministry of Science & Education. Azanza enjoys a doctoral grant from the Basque Government under the "Researchers Training Program".

## REFERENCES

[1] *5th Working IEEE / IFIP Conference on Software Architecture (WICSA 2005), 6-10 November 2005, Pittsburgh, Pennsylvania, USA*. IEEE Computer Society, 2005.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003.

[3] D. Batory. Program Refactoring, Program Synthesis, and Model Driven Development. In *Keynote at European Joint Conferences on Theory and Practice of Software (ETAPS) Compiler Construction Conference, Braga, Portugal, Mar 24 - Apr 1*, 2007.

[4] D. Batory, J. Liu, and J.N. Sarvela. Refinements and Multi-Dimensional Separation of Concerns. In *11th ACM Symposium on Foundations of Software Engineering (SIGSOFT) held jointly with 9th European Software Engineering Conference (ESEC/FSE), Helsinki, Finland, Sep 1-5*, 2003.

[5] D. Batory, J.Neal Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, June 2004.

[6] R. Capilla, F. Nava, S. Pérez, and J.C. Duenas. A Web-based Tool for Managing Architectural Design Decisions. In *Workshop on Sharing and Reusing Architectural Knowledge (SHARK 2006). ACM SIGSOFT Software Engineering Notes 31 (5). Co-located at the International Conference on Software Reuse (ICSR), 11-15 June, Torino, Italy.*, 2006.

[7] P. Clements, F. Bachman, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures. Views and Beyond*. Addison-Wesley, 2003.

[8] O. Díaz, S. Trujillo, and F. I. Anfurrutia. Supporting Production Strategies as Refinements of the Production Process. In *9th International Software Product Lines Conference (SPLC 2005), Rennes, France, September 26-29*, pages 210–221, 2005.

[9] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[10] D.S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, 2003.

[11] R. C. Holt, A. Winter, and A. Schürr. GXL: Toward a Standard Exchange Format. In *7th Working Conference on Reverse Engineering (WCRE 2000), Brisbane, Australia*, November 2000.

[12] A. G. J. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *5th IEEE/IFIP Working Conference on Software Architecture (WICSA)*, pages 109–119, November 2005.

[13] Lago P. van Vliet H. Kruchten, P. and T. Wolf. Building up and Exploiting Architectural Knowledge. In *5th IEEE/IFIP Working Conference on Software Architecture (WICSA)*, 2005.

[14] P. Kruchten. Architectural Blueprints. The 4+1 View Model of Software Architecture. *IEEE Software*, 12(6):42–50, 1995.

[15] P. Lago and P. Avgeriou, editors. *Reports First Workshop on Sharing and Reusing Architectural Knowledge (SHARK). ACM SIGSOFT Software Engineering Notes, 3(5), 32-36.*, 2006.

[16] J. Lee and D. Muthig. Feature-Oriented Variability Management in Product Line Engineering. *Communications of the ACM*, 49(12):55–59, December 2006.

[17] M. Lindvall, I. Rus, R. Jammalamadaka, and R. Thakker. Software Tools for Knowledge Management. Fraunhofer Center for Experimental Software Engineering, Maryland, USA, 2001. http://www.dacs.dtic.mil/techs/kmse/swtools4km.pdf.

[18] D.E. Perry and A.L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, pages 40–52, October 1992.

[19] M. Shaw and D. Garlan. *Software Architecture*. Prentice Hall, 1996.

[20] Sourceforge.net. GXL (Graph eXchange Language). http://gxl.sourceforge.net/.

[21] S. Trujillo. *Feature Oriented Model Driven Product Lines*. PhD thesis, School of Computer Sciences, University of the Basque Country, March 2007. http://www.struji.com.

[22] S. Trujillo, D. Batory, and O. Díaz. Feature Refactoring a Multi-Representation Program into a Product Line. In *5th International Conference on Generative Programming and Component Engineering (GPCE 2006), Portland, Oregon, USA, Oct 24-27*, 2006.

[23] S. Trujillo, D. Batory, and O. Díaz. Feature Oriented Model Driven Development: A Case Study for Portlets. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, Minnesota, USA, May 20-26*, 2007.

[24] J. Tyree and A. Akerman. Architecture decisions: Demystifying architecture. *IEEE Software*, 22(2):19–27, 2005.